Proceedings of the 2006 American Control Conference
Minneapolis, Minnesota, USA, June 14-16, 2006



# Scheduling and Sequence Reshuffle for Autonomous Aerial Refueling of Multiple UAVs

Zhipu Jin, Tal Shima, and Corey J. Schumacher

*Abstract*— **In this paper, we formulate the autonomous aerial refueling of multiple unmanned aerial vehicles (UAVs) as a scheduling problem. In order to find the optimal refueling sequence of UAVs, an efficient dynamic programming algorithm is introduced. When UAVs leave or join the queue, the optimal sequence needs to be recalculated. A systematic reshuffling method is developed such that the UAV sequence can be reconfigured by using the least amount of shuffle steps, where only one UAV changes its position in each step. By introducing a metric over UAV sequences, this reconfiguration effort is quantified and is treated as an additional cost which can be integrated into the dynamic programming algorithm.**

## I. INTRODUCTION

Unmanned Aerial Vehicles (UAVs) are commonly used in military and civilian applications. One of the limitations of many current UAVs is the restriction in flight duration due to the limited fuel capacity. Having autonomous aerial refueling (AAR) capability will allow UAVs to remain airborne longer and/or to take off with larger payload. Some work has already been done for modelling AAR docking maneuvers of a single UAV [1]. In this paper, we address the AAR problem as a scheduling problem in which a tanker needs to refuel multiple UAVs.

Considering the limited waiting time, finding the optimal refueling sequence for UAVs is similar to the scheduling problem for a single machine with "non-resumable" operations [2]. We assume that each UAV can only be refuelled once during the entire refueling process. Then, the problem is equal to driving the tanker to visit each UAV in some optimal order and it resembles in some aspects to the restricted travelling salesman problem and the vehicle routing problem with time windows [3], [4], [5]. Many efficient algorithms have been developed to solve these problems such as linear programming, branch-and-bound, and genetic algorithm. We use the dynamic programming method [6], [7] to develop an efficient algorithm to find the optimal sequence. By using a prior examination and feasibility tests during the execution, the proposed algorithm efficiently reduces the search space in cases where the constraints are active.

The optimal sequence needs to be recalculated whenever a UAV joins the queue or leaves it unexpectedly. We omit the dynamics of the UAVs and assume that the UAV sequence can be reconfigured by shuffling UAVs' location. The shuffling of cards, by subjecting a deck to a random permutation, is relatively well studied [8], [9], and a couple of shuffling algorithms were popularized by Knuth [10]. However, in the AAR problem we are interested in the minimum number of shuffling movements of UAVs to form a new, determined sequence. Moreover, we quantify the reconfiguration effort and integrate it into the dynamic programming algorithm.

The remainder of this paper is organized as follows. In section II, we formulate the scheduling problem for the AAR of multiple UAVs and an efficient dynamic programming algorithm is developed. We introduce a metric over sequences in section III to quantify the similarities among different UAV sequences. In section IV, three reshuffling algorithms are proposed for transferring one sequence into another. In section V, the effort of reconfiguration is quantified using the similarity metric as an additive cost. Conclusions are offered in section VI.

## II. AAR SCHEDULING PROBLEM OF MULTIPLE UAVS

In this section, we model the scheduling for the AAR of multiple UAVs as a combinatorial optimization problem. A tanker needs to provide refueling service for multiple UAVs. Each UAV has different parameters such as the current fuel level, refueling time, and the "Return-to-Field" priority. The last parameter, designated possibly by a human operator, indicates how important it is for a UAV to return for duty. The tanker gathers this information from all UAVs, decides the optimal refueling sequence, and sends the result back to each UAV. The UAVs form an echelon formation following the tanker according to the optimal sequence. We assume that communication between the tanker and the UAVs is ideal, i.e., information is sent between the tanker and UAVs without delays and errors. Thus, the problem we need to solve is a centralized optimization problem.

### A. Problem Formulation

Suppose that there are $N$ UAVs and each UAV is marked by an index $i$. The index set is $S = \{1, \cdots, N\}$. The parameters of each UAV are:

- Maximum waiting time $w_i$. This parameter indicates the longest time that the $i^{\text{th}}$ UAV can wait before refueling. The value of $w_i$ is changing w.r.t. time, i.e. it reduces

This work was supported by the Air Force Office of Scientific Research.

Z. Jin is with the Department of Electrical Engineering, California Institute of Technology, 1200 E California Blvd, Pasadena, CA 91125; IEEE Student Member; jzp@caltech.edu

T. Shima is with the Department of Aerospace Engineering, Technion - Israel Institute of Technology, Haifa 32000, Israel; this research was performed while the author held a National Research Council Research Associateship Award at the Control Design and Analysis Branch, US Air Force Research Labs, Wright-Patterson AFB, OH, 45433; IEEE Senior Member; tal.shima@technion.ac.il

C. Schumacher is a Research Aerospace Engineer, Control Design and Analysis Branch, Air Vehicles Directorate, Room 305, Building 146, Wright-Patterson AFB, OH, 45433; IEEE Senior Member; Corey.Schumacher@wpafb.af.mil

1-4244-0210-7/06/$20.00 ©2006 IEEE

as time progresses. We assume that $w_i > 0 \ \forall \ i \in S$ and that $w_i$ is sent, by each UAV, with a time stamp to enable synchronization.

- Refueling time $\tau_i$. This is the time that the tanker needs to fill up the $i^{\text{th}}$ UAV. It includes the time of docking maneuvers. In order to simplify the problem, we assume that $\tau_i$ is time-invariant and $0 < \underline{\tau} \leq \tau_i \leq \overline{\tau}$ for any $i \in S$.
- "Return-to-Field" priority $p_i$. This positive number is assigned to the $i^{\text{th}}$ UAV before it is sent for refueling. The larger $p_i$ is, the higher the UAV's priority is.
- Refueling sequence number $k \in S$. The tanker refuels UAVs according to this number from 1 to $N$.

For any refueling sequence, there exists a bijective function $f(\cdot) : S \to S$ such that $k = f(i)$ for any UAV $i$. The cost function for the AAR problem is defined as:

$$J = \sum_{i=1}^{N} \Big( p_i \cdot \sum_{k=1}^{f(i)} \tau_{f\ \ 1(k)} \Big), \tag{1}$$

where $\sum_{k=1}^{f(i)} \tau_{f\ \ 1(k)}$ is the total time needed for refueling the $i^{\text{th}}$ UAV and the UAVs before it in the queue. Suppose the set of all possible bijective functions is $F$. The optimal scheduling problem is finding the function $f(\cdot) \in F$ to minimize the cost function $J$. We can represent this as:

$$f(\cdot) = \arg \min_{f(\cdot) \in F} J \tag{2}$$

subject to:

$$w_i \geq \sum_{k=1}^{f(i)-1} \tau_{f\ \ 1(k)}, \ \forall i \in S. \tag{3}$$

Without the time constraints of inequations (3), there are totally $N!$ elements in $F$. However, the time constrains may make some of them unfeasible. Thus, the optimal scheduling problem is composed of two parts: (a) finding feasible sequences, and then (b) obtaining the optimal one. According to the formulation, the solution of equation (2) is not unique. For example, if two UAVs have the same parameters, then they can switch their position without affecting the cost. In that case, we do not distinguish between these solutions and just pick one heuristically, e.g., that with the smallest index number.

For the sake of simplicity, we assume in this paper that there always exists at least one feasible solution. In an actual implementation, if a feasible solution does not exist, it will be up to a human operator to decide which UAV can be sacrificed, and then the proposed algorithm can be re-run.

### *B. Dynamic Programming Algorithm*

In order to develop the search algorithm, a layered structure with $N+2$ layers of nodes is introduced. Each layer is marked by an index $j \in \{0, 1, \cdots, N+1\}$ which corresponds to one stage in dynamic programming. The nodes in each layer represent UAVs that may be refueled at that stage. We use $i_j \in S$ to indicate these nodes except on the initial layer $(j = 0)$ where there is only one virtual starting node $i_0 = 0$ and on the final layer $(j = N + 1)$ which only includes one sink node $i_{N+1} = -1$. The nodes set on each layer is defined by $S_j \subseteq S$. The scheduling problem is to find an optimal path $\pi(0, -1)$ from the starting node to the sink node by connecting nodes on adjacent layers. For each layer, only one node can be visited. Also, each node can be visited only once. When the path is found, the function $f(\cdot)$ is determined.

For each layer, the node set $S_j$ is formed according to a prior examination. For node $i \in S$, if there exists a subset $K \subseteq S \setminus \{i\}$ and $|K| = j - 1$ such that

$$w_i \geq \sum_{n \in K} \tau_n, \tag{4}$$

then $i \in S_j$. $|K|$ is the number of elements in set $K$. This prior examination is important when the time constraints are tight.

Following are two lemmas that are easy to prove according to the above definition of the layer structure.

*Lemma 2.1:* If there exists a feasible path in the layer structure, then $|S_j| \geq N + 1 - j$ for any $j \in \{1, 2, \cdots, N\}$.

*Lemma 2.2:* For any $j \in \{1, \cdots, N-1\}$, $S_{j+1} \subseteq S_j$.

When we construct the layer structure, we determine $S_N$ first, then find $S_{N-1}$ by joining $S_N$ and the examination result over $S \setminus S_N$, then find $S_{N-2}$, and so forth. After constructing the layer structure, we break the problem into $N$ stages and define $T(0, -1)$ as the cost of the optimal path from the initial node to the sink node. The $N$ stages correspond to the $N$ layers, excluding the initial and final one.

Before the path reaches the sink node, it must reach a node $i_N \in S_N$. Therefore,

$$T(0, -1) = \min_{i_N \in S_N} \Big( T\big(0, i_N\big) + d(i_N, -1) \Big) \tag{5}$$

where $d(i_N, -1)$ is the cost from $i_N$ to the sink node and it equals $p_{i_N} \tau_{i_N}$. For any other stage $j$, given the sequence $\{i_j, \cdots, i_N\}$, we have the similar recursion equation:

$$T(0, i_j) = \min_{i_j \ \ 1 \in S_j \ \ 1 \setminus \{i_j, \cdots, i_N\}} \big( T(0, i_{j-1}) + d(i_{j-1}, i_j) \big) \tag{6}$$

where the cost $d(i_{j-1}, i_j)$ can be calculated by

$$d(i_{j-1}, i_j) = \tau_{i_j \ \ 1} \sum_{n=j-1}^{N} p_{i_n}. \tag{7}$$

For the initial layer, the recursion is:

$$T(0, i_1) = d(0, i_1) \tag{8}$$

where $d(0, i_1) = 0$.

At each stage, an additional feasibility test is needed. Let $\Gamma = \sum_{i=1}^{N} \tau_i$ be the total refueling time for all UAVs. At stage $(j - 1)$, a feasibility test for node $i_{j-1} \in S_{j-1} \setminus \{i_j, \cdots, i_N\}$ is that if

$$w_{i_j \ \ 1} \geq \Gamma - \sum_{n=j-1}^{N} \tau_{i_n}, \tag{9}$$

then $i_{j-1}$ is feasible. If there is no node that passes this test, we let $T(0, i_j)$ be big enough such that it cannot be selected at the pervious stage. One reasonable large value is:

$$T(0, i_j) = N \cdot \Gamma \cdot \max(p_i). \tag{10}$$

Given $\{i_j, \cdots, i_N\}$, the nodes on layer $(j-1)$ that pass the test compose the feasible set $\Omega_{j-1}$. When the algorithm finishes searching, if $T(0,-1) \geq N \cdot \Gamma \cdot \max(p_i)$, it means that there does not exist a feasible refueling sequence to meet the time constraints.

The computation complexity is sensitive to the time constraints. In the worst case, the time constraints are satisfied by any permutation of $S$ and the scheduling problem is solved in time $O(N^2 2^N)$ as discussed in [6]. The easiest case is when there exists only one feasible sequence that can meet the time constraints. Then, as soon as the layer structure is determined, the optimal sequence is found.

The dynamic programming algorithm described above can be used to solve general AAR scheduling problems. Moreover, according to the structure of the cost function in equation (1), we have two rules that greatly reduce the computation time.

*Proposition 2.3:* Suppose at stage $j-1$, $\Omega_{j-1}$ is the feasible set of layer $j-1$ for the given sequence $\{i_j, \cdots, i_N\}$. For any $m$, $n \in \Omega_{j-1}$, if $\tau_m = \tau_n$ and $p_m < p_n$, then

$$T(0, m) + d(m, i_j) < T(0, n) + d(n, i_j). \tag{11}$$

*Proposition 2.4:* Suppose at stage $j-1$, $\Omega_{j-1}$ is the feasible set of layer $j-1$ for the given sequence $\{i_j, \cdots, i_N\}$. For any $m$, $n \in \Omega_{j-1}$, if $p_m = p_n$ and $\tau_m < \tau_n$, then

$$T(0, m) + d(m, i_j) > T(0, n) + d(n, i_j). \tag{12}$$

According to these propositions, in each recursive step starting from the end of the queue and moving forward, we pick the node with the least priority from those feasible nodes with the same refueling time, or pick the node with the largest refueling time from those with the same priority. These two propositions can reduce the complexity of the scheduling problem.

Following is an example showing how the dynamic programming algorithm works. Suppose there are 4 UAVs waiting for refuelling. Table I lists all the parameters of the problem and Table II is the layer structure. The nodes in each column compose the feasible node set $S_j$. Those nodes in bold form the optimal sequence. Node 3 is the only one in layer 4, so it must be selected. Then at layer 3, node 1 and 4 have the same refueling time. According to Proposition 2.3, node 1 is selected. After comparing the sequences $\{2, 4\}$ and $\{4, 2\}$, we obtain the optimal sequence as $\{4, 2, 1, 3\}$ with the cost of 98.

TABLE I

PARAMETERS OF UAVS IN THE REFUELING SEQUENCE.

| UVA index $i$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Max. waiting time $w_i$ | 14 | 6 | 22 | 12 |
| Refueling time $\tau_i$ | 5 | 6 | 4 | 5 |
| Priorities $p_i$ | 2 | 1 | 2 | 3 |

TABLE II

LAYER STRUCTURE OF INITIAL SCHEDULING.

| Layer 0 | Layer 1 | Layer 2 | Layer 3 | Layer 4 | Layer 5 |
|---|---|---|---|---|---|
| **0** | 1 | 1 | **1** | **3** | **-1** |
| | 2 | **2** | 3 | | |
| | 3 | 3 | 4 | | |
| | **4** | 4 | | | |

## III. SIMILARITY METRIC BETWEEN UAV SEQUENCES

For the AAR problem, the number of UAVs may change from time to time. We assume that UAVs do not join or leave the queue simultaneously and the interval between any two arrivals or departures is long enough such that the new echelon formation is already formed before the next UAV joins or leaves. In this section, we focus on how to rearrange the sequence when a new UAV joins. We assume that, at first step, the new UAV is appended to the end of the echelon formation. Then, after the new optimal sequence is found, the formation is reconfigured accordingly. Intuitively, the similar the new optimal sequence is to the old one, the less reconfiguration is needed.

A metric is defined to quantify the similarity between two sequences that have the same nodes. Suppose there is a node set $M$ which has $N$ nodes. A permutation group is a sequence group $G$ whose elements are all permutation sequences of $M$. Any element $x \in G$ is a sequence with $N$ nodes. For each node $e_i \in M$ in the sequence $x = [e_1, e_2, \cdots, e_N]$ there exists two adjacent nodes $(e_i^l = e_{i-1}, e_i^r = e_{i+1})$. For the first node $e_1$ and the last node $e_N$, the adjacent node pairs are $(e_1^l = \varnothing, e_1^r = e_2)$ and $(e_N^l = e_{N-1}, e_N^r = \varnothing)$, respectively, where $\varnothing$ means "None". For any element $x \in G$, the adjacent node pair of node $e_i$ is $(x(e_i)^l, x(e_i)^r)$. For any $x1$, $x2 \in G$, we assume that the set $k_1$ is composed of the nodes that have identical neighbors; the set $k_2$ is composed of the nodes that only have the same left neighbors; the set $k_3$ is composed of the nodes that only have the same right neighbors; and the set $k_4$ is composed of the nodes that have different neighbors. It is clear that $|k_1| + |k_2| + |k_3| + |k_4| = N$.

For a permutation group $G$ and node set $M$, suppose $x1$, $x2 \in G$, a metric $\mathcal{D}(x1, x2)$ is defined as

$$\mathcal{D}(x1, x2) = \sum_{i=1}^{N} \mathcal{E}(e_i) \tag{13}$$

where $e_i \in M$ and

$$\mathcal{E}(e_i) = \begin{cases} 2, & \text{if } x1(e_i)^l \neq x2(e_i)^l \text{ and } x1(e_i)^r \neq x2(e_i)^r \\ 1, & \text{if } x1(e_i)^l \neq x2(e_i)^l \text{ and } x1(e_i)^r = x2(e_i)^r \\ 1, & \text{if } x1(e_i)^l = x2(e_i)^l \text{ and } x1(e_i)^r \neq x2(e_i)^r \\ 0, & \text{otherwise} \end{cases} \tag{14}$$

For example, suppose $M = \{1, 2, 3, 4, 5\}$, and $G$ has $5! = 120$ elements. Let $x1 = [1, 3, 4, 5, 2]$ and $x2 = [3, 4, 5, 1, 2]$, then $\mathcal{D}(x1, x2) = 5$ since both neighbors of node 1 are

changed, the left neighbors of 3 and 2 are changed and the right neighbor of 5 is changed.

We now discuss some of the general properties of this metric. According to the definition, $\mathcal{D}(x1, x2) = 0$ if and only if $x1 = x2$. Also, $\mathcal{D}(x1, x2) = \mathcal{D}(x2, x1)$. The next lemma shows that $\mathcal{D}$ satisfies the triangle inequality.

*Lemma 3.1:* For any $x1$, $x2$, and $x3 \in G$, $\mathcal{D}(x1, x2) + \mathcal{D}(x2, x3) \geq \mathcal{D}(x1, x3)$

*Proof:* Suppose that, for $x1$ and $x2$, there exist four node sets as $k_1$, $k_2$, $k_3$, and $k_4$ that are defined before. So,

$$\mathcal{D}(x1, x2) = |k_2| + |k_3| + 2|k_4|. \tag{15}$$

For $x2$ and $x3$, there exist the similar node sets as $\hat{k}_1$, $\hat{k}_2$, $\hat{k}_3$, and $\hat{k}_4$. Thus,

$$\mathcal{D}(x2, x3) = |\hat{k}_2| + |\hat{k}_3| + 2|\hat{k}_4|. \tag{16}$$

Now suppose that the intersection of $k_1$ and $\hat{k}_1$ has $r$ nodes, then between $x1$ and $x3$, there are at least $r$ nodes that have identical neighbors. We rewrite $k_1$ and $\hat{k}_1$ as

$$\begin{cases} |k_1| &= r + \alpha \\ |\hat{k}_1| &= r + \beta. \end{cases} \tag{17}$$

Note that these $\alpha$ nodes must belong to $\hat{k}_2 \bigcup \hat{k}_3 \bigcup \hat{k}_4$. Since nodes in $\hat{k}_2$, $\hat{k}_3$, or $\hat{k}_4$ make different contributions to $\mathcal{D}$, we assume that there are $n1$ nodes in $\alpha$ that belong to $\hat{k}_2 \bigcup \hat{k}_3$ and $n2$ nodes that belong to $\hat{k}_4$. Thus, we have

$$\begin{aligned} \alpha &= n1 + n2 \\ \mathcal{D}(x1, x2) &= n1 + 2 \cdot n2 + \Psi_1. \end{aligned} \tag{18}$$

For the same reasons, we have

$$\begin{aligned} \beta &= m1 + m2 \\ \mathcal{D}(x2, x3) &= m1 + 2 \cdot m2 + \Psi_2. \end{aligned} \tag{19}$$

For $x1$ and $x3$, we have

$$\begin{aligned} \mathcal{D}(x1, x3) &\leq n1 + 2 \cdot n2 + m1 + 2 \cdot m2 + \Psi_1 + \Psi_2 \\ &= \mathcal{D}(x1, x2) + \mathcal{D}(x2, x3). \end{aligned} \tag{20}$$

■

*Lemma 3.2:* If $x1$, $x2 \in G$ and $x1 \neq x2$, then $4 \leq \mathcal{D}(x1, x2) \leq 2N$.

The previous lemma can be proved easily according to the metric definition.

## IV. Transfer between Sequences

Transferring a sequence to another one, by using efficient shuffle steps, is the main topic of this section. The answer directly affects the echelon formation sequence reconfiguration, whenever a UAV joins or unexpectedly leaves the refueling queue. Due to the expected severe flight safety requirements near the tanker we assume that the reshuffling is performed for one UAV at a time.

The single-node shuffle is defined as:

*Definition 4.1:* A single-node shuffle for any element of the sequence $x$ in the permutation group $G$, is transferring one node from its position in the sequence to a different one, while the ordering of the other nodes is unchanged.

For example, we have a sequence with five nodes as $x1 = [1, 2, 3, 4, 5]$. Moving node 4 to the position between 1 and 2 results with a new sequence $x2 = [1, 4, 2, 3, 5]$. Multiple single-node shuffle steps may be needed for a sequence transformation. Thus, for any $x1$, $x2 \in G$, an efficient reshuffle algorithm generates a sequence of single-node shuffle steps such that, by implementing these shuffle steps, $x1$ can be transferred into $x2$.

### A. Reshuffle algorithm one

Suppose the initial sequence is $x1 = [a_1, a_2, \cdots, a_N]$ and the final sequence is $x2 = [b_1, b_2, \cdots, b_N]$. Reshuffle algorithm one is:

- Let k=1 and $\hat{x} = x1$;
- From the $k^{\text{th}}$ node in $\hat{x}$, from left to right, find the node $a_i$ in $\hat{x}$ such that $a_i = b_k$. If $a_k = b_k$, keep $\hat{x}$ and directly jump to the next step. Otherwise, implement a single-node shuffle by moving $a_i$ to the $k^{\text{th}}$ place and generate a new $\hat{x}$, then go to next step.
- Let k=k+1 and repeat the previous step until $k = N$.

It is easy to show that, in the worst case, this algorithm needs $N - 1$ single-node shuffle steps and $N(N-1)/2$ comparisons to transfer $x1$ into $x2$. The disadvantage of this algorithm is that it cannot guarantee to find the minimum single-node shuffle steps for a sequence transformation. For example, suppose $x1 = [1, 2, 3, 4, 5]$ and $x2 = [2, 3, 4, 5, 1]$. The algorithm needs four shuffle steps to transfer $x1$ to $x2$. Obviously, the minimum number of single-node shuffle step is one by moving node 1 to the right side of node 5.

### B. Reshuffle algorithm two

In order to find a better reshuffle algorithm, we introduce the concept of subsequence. For $x1$, $x2 \in G$, there exists a subsequence partition such that each element $\delta$ in this partition is the largest non-empty subsequence in which the nodes keep the same order for $x1$ and $x2$. For example, suppose $x1 = [1, 2, 3, 4, 5]$ and $x2 = [3, 4, 5, 1, 2]$. It is easy to see that $\{[1, 2], [3, 4, 5]\}$ is the subsequence partition of $x1$ and $x2$. By switching the position of these two subsequences, $x1$ can be transferred into $x2$. We define a subsequence shuffle as:

*Definition 4.2:* A subsequence shuffle of sequence $x$ is moving a subsequence $\delta$ to a different location which is composed by $|\delta|$ single-node shuffles. The node order inside $\delta$ is not changed.

The subsequence shuffle resembles shuffling a deck of cards by moving multiple cards together. The single-node shuffle is a specific case of the subsequence shuffle. Thus, a sequence transformation can be treated into two levels: subsequence level and node level.

For a sequence transformation, it is important to find the subsequence partition. Similar to the single node, we define the left subsequence neighbor of subsequence $\delta$ in $x$ as $x(\delta)^l$ and the right subsequence as $x(\delta)^r$. All the elements of the partition can be put into three subsequence sets $\Lambda_1$, $\Lambda_2$, and $\Lambda_3$. Elements in $\Lambda_1$ only have the same left subsequence neighbors in $x1$ and $x2$. Elements in $\Lambda_2$ only have the same

right subsequence neighbors. Elements in $\Lambda_3$ do not have any same subsequence neighbor in $x1$ and $x2$. Finding $\Lambda_1$, $\Lambda_2$, and $\Lambda_3$ can be done systematically. Suppose we already have $k_1$, $k_2$, $k_3$, and $k_4$ which are defined in Section III. Let us start from the nodes of $k_1$. Suppose node $a \in k_1$, then subsequence $\delta = [x1(a)^l, e, x1(a)^r]$ does not change its node order in the transformation from $x1$ to $x2$. If $x1(e)^l \in k_1$, then $\delta$ extends by adding $x1(e)^l$'s left neighbor on its left side until this left neighbor is $\varnothing$. If $x1(e)^l \in k_3$, then $\delta$ cannot extend itself on the left side. The same extending process can be done for $x1(e)^r$. Eventually, $\delta$ is extended to be the largest subsequence including $a$. During this process, the nodes used to form $\delta$ are eliminated from $k_1$, $k_2$, and $k_3$. When $k_1 = \emptyset$, we check any single-node subsequence in $k_2$. If a node $b$ belongs to $k_2$ and its left neighbor $x1(b)^l \neq \varnothing$, then $x1(b)^l$ must belong to $k_3$ since $k_1 = \emptyset$. Thus, $[x1(n)^l, n]$ is formed and belongs to $\Lambda_3$. After the extending processes, we can find all the elements of $\Lambda_1$, $\Lambda_2$, and $\Lambda_3$.

*Lemma 4.3:* $|\Lambda_1| \leq 1$ and $|\Lambda_2| \leq 1$.

*Proof:* The only possible element in $\Lambda_1$ is the largest subsequence that locates on the first position from left in $x1$ and $x2$. The same result holds for $\Lambda_2$. ■

Suppose $\Lambda_1$, $\Lambda_2$, and $\Lambda_3$ are subsequences sets for the sequence transformation from $x1$ to $x2$. Reshuffle algorithm two is:

- Find the smallest subsequence $\delta$ in $\Lambda_3$ with the condition that no node in $\delta$ has not been moved before.
- According to $x2$, find the left subsequence $x2(\delta)^l$ and the right subsequence $x2(\delta)^r$;
- Implement a subsequence shuffle step such that
  - If $x2(\delta)^l \in \Lambda_1$ or if $x2(\delta)^l \notin \Lambda_1$ and $|x2(\delta)^l| \geq |x2(\delta)^r|$, then put $\delta$ on the right side of $x2(\delta)^l$ to form a new subsequence.
  - If $x2(\delta)^r \in \Lambda_2$ or if $x2(\delta)^r \notin \Lambda_2$ and $|x2(\delta)^l| < |x2(\delta)^r|$, then put $\delta$ on the left side of $x2(\delta)^r$ to form a new subsequence.
- Update $\Lambda_1$, $\Lambda_2$, and $\Lambda_3$ according to these new subsequences.
- Repeat the previous steps until $\Lambda_3$ is empty;

For reshuffle algorithm two, the most important part is moving subsequences in $\Lambda_3$ to generate longer subsequences and to reduce the number of elements of $\Lambda_3$. Whenever one subsequence $\delta$ is moved, the number of elements of $\Lambda_3$ is decreased at least by one. This algorithm can be executed in polynomial time. However, it still cannot guarantee to find the minimum number of single-node steps.

### C. Reshuffle algorithm three

By using the principle of optimality, we develop the reshuffle algorithm three to find the minimum number of single-node steps.

Suppose $\Lambda_1$, $\Lambda_2$, and $\Lambda_3$ are subsequences sets for $x1$ and $x2$ where $\Lambda_3$ has $m$ elements. The minimum single-node shuffle steps we need is represented by $\mathcal{T}(m)$. We have

$$\mathcal{T}(m) = \min_{\delta_i \in \Lambda_3} \left( \min(\mathcal{T}(\hat{m})^l + |\delta_i|, \mathcal{T}(\hat{m})^r + |\delta_i|) \right) \quad (21)$$

where $\mathcal{T}(\hat{m})^l$ is the minimum single-node shuffle steps we need after $\delta_i$ is moved to the right side of its left subsequence neighbor, and the same for $\mathcal{T}(\hat{m})^r$. The subsequence set $\Lambda_3$ needs to be updated at each recursive step. Let $\hat{m}$ represent the size of $\Lambda_3$ after updating. Sometimes, moving $\delta_i$ to the right side of its left neighbor is also connecting it to its right neighbor. In that case, these three subsequences are formed into one larger subsequence and the elements number of $\Lambda_3$ is reduced by two. This recursive algorithm stops when $\hat{m} = 0$.

By using reshuffle algorithm three, $\mathcal{T}(m)$ is guaranteed to be the minimum number of single-node shuffles. The computation time of this recursive algorithm depends on $m$, i.e., how many elements exist in $\Lambda_3$. In the worst case, the results are found in time $O(m!)$.

### D. Comparison

In order to verify the feasibility of these algorithms, we tested them on permutation groups with different number of nodes. The algorithms were coded in Matlab and run on a desktop with a Xeon(TM) CPU at 2.66 GHz and 1.00 GB of RAM. For each permutation group, we randomly picked one sequence as the initial sequence and calculated the shuffle steps that transfer this initial one to all the other permutations. The average value of shuffle steps and computation time are calculated over each permutation group. Fig. 1 and Fig. 2 show the average values of shuffle steps and computation time for the three different algorithms respectively. According to the simulation results, algorithm three guarantees the least amount of shuffle steps for a sequence transformation, but it needs much more computation time to find solutions than other two algorithms.

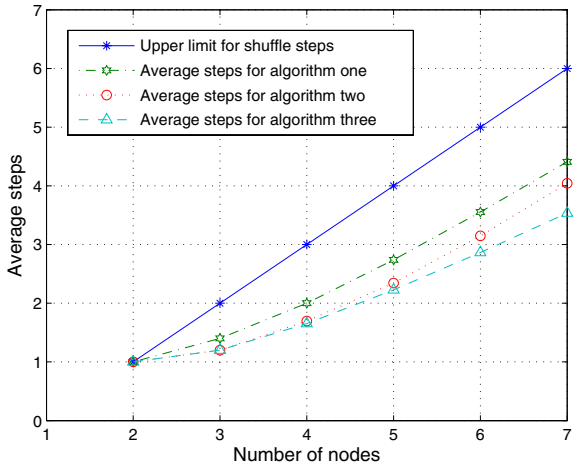


Fig. 1. Average number of shuffle steps for different algorithms

## V. Sequence Reconfiguration for AAR

We assume that the reconfiguration of the UAV echelon formation is performed by shuffling the location of one UAV at each time. This method can naturally ease the collision avoidance issue. The cost of the reconfiguration is related to

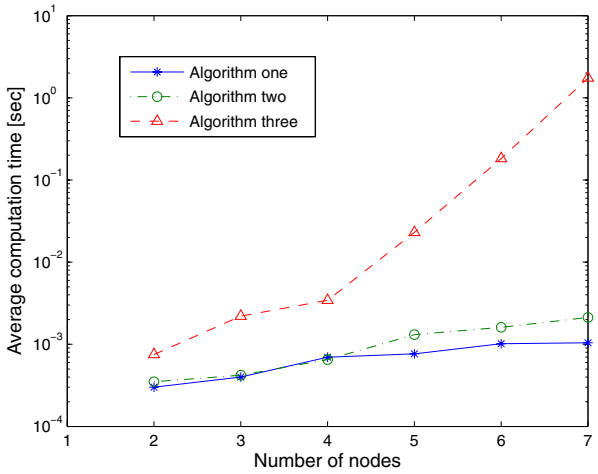


Fig. 2. Average computation time for different algorithms

how many shuffle steps are needed. So far, we can calculate the shuffle steps using the algorithms of section IV only after the new sequence is determined. This means that we cannot directly consider the number of shuffle steps into the dynamic programming algorithm of section II. However, in order to avoid the time-consuming exhaustive searching method, we try to find another description of the reconfiguration cost which can also fit into the dynamic programming method. Intuitively, the more similar two sequences are, the less shuffle steps are needed. The metric $\mathcal{D}$ defined in section III can represent this similarity. Since the metric $\mathcal{D}$ is an additive function over the nodes, it can be easily integrated into the dynamic programming algorithm. Thus, we choose the metric $\mathcal{D}$ to indicate the cost of the reconfiguration.

Suppose the initial refueling sequence is $[i_1, \cdots, i_N]$. For each node $i_j$, the two adjacent nodes are $(i_j^l = i_{j-1}, i_j^r = i_{j+1})$. When a new UAV joins the queue, we assume that a new node $i_{N+1}$ is appended to the end of the queue and $\pi_o = [i_1, \cdots, i_N, i_{N+1}]$. The new optimal sequence is indicated by $\pi_n$. We redefine the total cost function for refueling scheduling as:

$$\begin{aligned} J &= K_1 \sum_{i=1}^{N+1} \left( p_i \cdot \sum_{k=1}^{f_n(i)} \tau_{f_n{}^1(k)} \right) + K_2 \mathcal{D}(\pi_o, \pi_n) \\ &= K_1 \sum_{i=1}^{N+1} \left( p_i \cdot \sum_{k=1}^{f_n(i)} \tau_{f_n{}^1(k)} \right) + K_2 \sum_{j=1}^{N+1} \mathcal{E}(i_j) \end{aligned} \quad (22)$$

where $f_n(\cdot)$ is the new scheduling mapping function, the second term represents the metric distance, and $(K_1, K_2)$ are the weight coefficients. Also, there are $N+1$ time constraints listed below:

$$w_i \geq \sum_{k=1}^{f_n(i)-1} \tau_{f_n{}^1(k)}, \forall i \in \{1, \cdots, N+1\}. \quad (23)$$

The additive property of the new cost function makes the dynamic programming algorithm in section II still effective. The cost $d(i_{j-1}, i_j)$ in each recursive step is calculated by

$$d(i_{j-1}, i_j) = K_1 \cdot \tau_{i_j \ 1} \cdot \sum_{n=j-1}^{N} p_{i_n} + K_2 \cdot \mathcal{E}(i_j) \quad (24)$$

and $d(0, i_1) = K_2 \cdot \mathcal{E}(i_1)$.

## VI. CONCLUSIONS

In this paper, a dynamic programming algorithm was developed for the AAR scheduling problem. In this problem one tanker needs to refuel multiple UAVs flying in an echelon formation. The optimal sequence is based on the UAVs parameters, including time constraints. When refueling time constraints are tight, a prior examination and feasibility tests in each recursive step are necessary to reduce the search space and thus make the search more efficient.

When a UAV joins, or leaves the queue unexpectedly, the optimal sequence needs to be recalculated. We introduced a metric to indicate how similar the new sequence is to the old one and chose it as the reconfiguration cost. The additive property of the metric makes it possible to add it to the dynamic programming algorithm as an additional cost term. Efficient algorithms for the reshuffling have also been proposed, including a computationally intensive one that provides the minimum number of shuffle steps.

## ACKNOWLEDGMENTS

The authors would like to thank Prof. Kevin Passino from The Ohio State University for helpful discussions.

## REFERENCES

[1] M. L. Fravolini, A. Ficola, G. Campa, M. R. Napolitano, and B. Seanor, "Modeling and control issues for autonomous aerial refueling for uavs using a probe-drogue refueling system," *Aerospace Science and Technology*, vol. 8, pp. 611–618, 2004.

[2] P. Mauguiere, J. C. Billaut, and J. L. Bouquard, "New single machine and job-shop scheduling problems with availability constraints," *Journal of Scheduling*, vol. 8, no. 3, pp. 211–231, June 2005.

[3] Y. Dumas, J. Desrosiers, E. Gelinas, and M. M. Solomon, "An optimal algorithm for the traveling salesman problem with time windows," *Operations Research*, vol. 43, no. 2, pp. 367–371, March-April 1995.

[4] E. Balas and N. Simonetti, "Linear time dynamic-programming algorithms for new classes of restricted TSPs: A computational study," *INFORMS Journal on Computing*, vol. 13, no. 1, pp. 56–75, WIN 2001.

[5] M. Desrochers, J. Desrosiers, and M. Solomon, "A new optimization algorithm for the vehicle routing problem with time windows," *Operations Research*, vol. 40, no. 2, pp. 342–354, March-April 1992.

[6] R. Bellman, "Dynamic programming treatment of the travelling salesman problem," *Journal of the ACM*, vol. 9, no. 1, pp. 61–63, Jan. 1962.

[7] D. K. Smith, *Dynamicd Programming : A Practical Introduction*, ser. Mathematics and its Applications. Ellis Horwood Limited, 1991.

[8] D. Bayer and P. Diaconis, "Trailing the dovetail shuffle to its lair," *The Annals of Applied Probability*, vol. 2, no. 2, pp. 294–313, 1992.

[9] L. N. Trefethen and L. M. Trefethen, "How many shuffles to randomize a deck of cards," *Proceedings of the Royal Society Londan A*, vol. 456, pp. 2561–2568, 2000.

[10] D. E. Knuth, *The art of computer programming*, ser. Addison-wesley series in computer science and information processing. Addison-wesley, 1981, vol. 2.

[11] M. Savelsbergh, "Local search in routing problem with time windows," *Annals of Operations Research*, vol. 4, pp. 285–305, 1986.